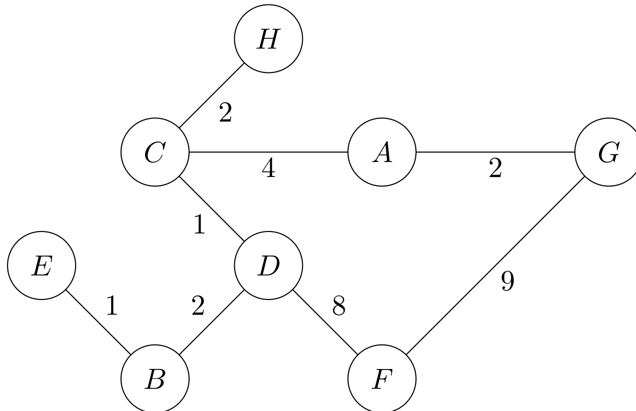
CSM 61B Fall 2020

Graph Algorithms

Mentoring 10: October 24-30, 2020

1 Searches

1.1 For the graph below, write the order in which vertices are visited using the specified algorithm starting from A. Break ties by alphabetical order. Notice that we have now introduced edge weights to the graph.



(a) DFS

$A - C - D - B - E - F - G - H$

(b) BFS

$A - C - G - D - H - F - B - E$

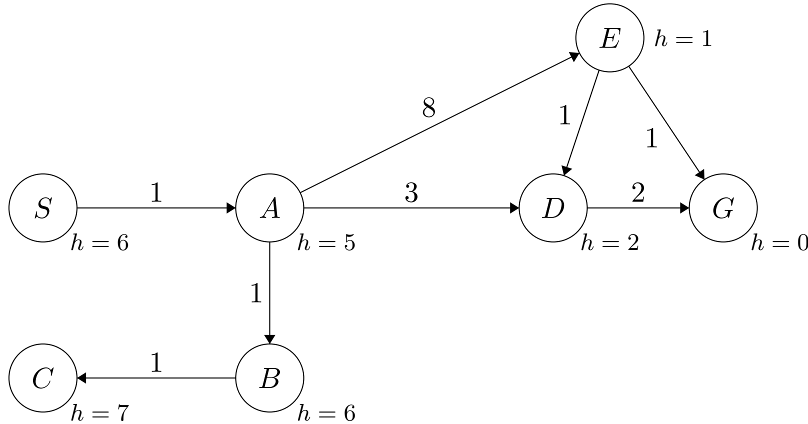
(c) Dijkstra's

$A - G - C - D - H - B - E - F$

2 Shortest Paths

2.1 Find the path from the start, S, to the goal, G, when running each of the following algorithms.

The **heuristic**, h, estimates the distance from each node to the goal.



(a) Which path does Dijkstra's return?

$S - A - D - G$

From the starting node, choose the path that has the least cost, go to that node, and repeat until we reach the goal node. We choose the lowest *total cost*.

We keep a priority-queue fringe that keeps track of paths. At each step, we remove the shortest path from the fringe and add its children to the fringe, trying all paths in increasing cost order until we reach G.

(b) Which path does A* search return?

A* search is an algorithm that combines the total distance from the start with the heuristic to optimize the search procedure.

$S - A - D - G$

At each node, we choose the next node that has the lowest sum of the path cost and $h(\cdot)$ value. This is essentially uniform cost search and greedy search combined.

For A* to work, heuristics must be *admissible* and *consistent*.

- Admissible heuristics underestimate the true distance to the goal.
- Consistent heuristics require that the difference in heuristic values between two nodes cannot be greater than the true distance between the two.

(c) What is the runtime of Dijkstra's? A*? What is the space requirement

for both?

We assume a binary heap Priority Queue. The largest the PQ can ever get is size V since there are V vertices, and we never add vertices twice. (We update using `decrementKey` instead.)

In the worst case, we do the following in Dijkstra's:

- Insert every vertex into the PQ (V vertexes, $O(\log V)$ time)
- Remove every vertex from the PQ (V vertexes, $O(\log V)$ time)
- Update every vertex in the PQ (E edges, $O(\log V)$ time)

Hence, Dijkstra's has runtime $O(V \log V + V \log V + E \log V) = O(E \log V)$ since $E > V$.

A* has the same runtime as Dijkstra's in the worst case. We can see this by constructing a very poor heuristic that returns 0 for all vertices! We can see that this heuristic is trivially admissible (distance to the goal is at least 0, so it must be admisible) and trivially consistent (the difference is always 0, which is not greater than the true distance between any two nodes). Then, the behaviour of A* on the graph is exactly like Dijkstra's.

However, given a good heuristic, A* can have a better average runtime, which is why we often prefer it.

The space requirement for the graph is $\Theta(V+E)$ assuming an adjacency list, and the space for the priority queue is $\Theta(V)$;

3 True and False

3.1 State if the following statements are True or False, and justify. For all graphs, assume that edge weights are positive and distinct, unless otherwise stated.

(a) Adding some positive constant k to every edge weight does not change the shortest path tree from vertex S.

False. A counterexample can be thought of as follows: take a triangular graph with 3 vertices A, B, and C and 3 edges: A-B with weight 1, B-C with weight 2, and A-C with weight 5. We would like to start at A and end at C. The original shortest path from A –¿ C involves passing through B, but after adding a constant of 3, the shortest path is now directly taking the edge from A to C.

(b) Doubling every edge weight does not change the shortest path tree.

True. Doubling the weight of every edge, the equivalent of multiplying by 2, applies the exact same transformation to every edge. This means the relationship between the edges remains constant the same edges will be selected and the final cost of the shortest path will be doubled.

Following from the example above if we have a graph with edge A→ B (w = 1), B→ C (w = 2), and A→C(w =5) the shortest path will be A → B → C. However when we double each edge weight we have A→ B (w = 2), B→ C (w = 4), and A→C(w =10) and run disjkstras again from A → C our shortest path will be A → B → C with a cost of 8 compared to A → C at a cost of 10. Multiplying our edges by values ensures that they maintain the same ratio and the shortest path will not change.

(c) If the weight of each edge is decreased by 1, then the resulting shortest path in any graph from u to v is unchanged.

False.
The effect of adding/subtracting a constant to/from each edge depends on the number of edges in a path. Subtracting 1 from every edge makes paths with more edges shorter. Subtracting from an edge can also make it negative.

(d) If an edge e is the lightest edge connected to vertex S, it must be a part of the shortest path tree from vertex S.

True.
Starting from vertex S, you will always choose the lightest edge connected to vertex S.

(e) Consider a graph G, where every edge is nonnegative, except the edges adjacent to vertex s. Dijkstra's usually fails on graphs with negative edge weights, however if we run Dijkstra's starting from s, we will get the correct shortest paths tree.

True.
Dijkstra's fails if incorporating a negative edge not yet seen decreases the shortest path. In the case, all negative edges have been seen and added to the fringe. That means adding more edges to any forming path can only increase the total distance (since all other edge weights are nonnegative.